Solving partial differential equations with neural networks
FYS-STK3155 - Project 3

Gulla Serville Torvund and Karl Henrik Fredly
(Dated: December 16, 2020)

In this project, we will look at the application of neural networks to solving partial differential equations. The central idea is to let the neural network model the solution to the equation, and to train the network by fitting it to the differential equation with gradient descent and automatic differentiation. We find that the neural network is able to solve the diffusion equation in one dimension, but is outclassed by the simple explicit scheme. The neural network achieves a mean absolute error of 7e-04, while the explicit scheme achieves a mean absolute error of 2e-05 with little effort. We also solve an equation which gives us the largest and smallest eigenvalues of a symmetric matrix. The performance of the neural network depends on the matrix in question, but achieves an error of 3e-05 at best after a 15 minute calculation and some luck. Traditional eigenvalue solvers completely outclass the network, but these applications show some of the great flexibility of neural networks.

I. INTRODUCTION

More or less every fundamental law of nature are partial differential equations (PDEs), as they depend on various rates of change. Many PDEs are complicated and impossible to solve analytically, and scientists have therefore for a long time turned to the computer for help. With the entrance of deep learning, artificial intelligence has also proved to be good at solving PDEs.

As aspiring physicists, we want to explore the way we can use neural networks to solve PDEs. We will in this project deal with simple PDEs so that we can compare the performance of our neural network with the analytical solution.

In the methods section (II) we will look closer at the problems we are dealing with in this project, which are the diffusion equation and the finding of the largest and smallest eigenvalues of a symmetric matrix. We will explain the explicit scheme algorithm and the structure of our neural network, which are the methods we use for solving the partial differential equations.

We present our results in section III, including the performance of the neural network on both the diffusion equation and the eigenvalue problem.

In section IV we discuss the results, how well the neural network performs, and what limitations it has.

Finally, we will come to a conclusion and summarize the project in section V.

All code used to calculate the results in this report can be found here: https://github.com/KarlHenrik/FYS-STK-Gruppe/tree/master/Project3. For this project we implemented the explicit scheme for solving the diffusion equation and wrote classes which uses the sequential model in TensorFlow/Keras together with the tensorflow.GradientTape API used to solve the diffusion equation and the equation used in the eigenvalue problem. A short example of how tensorflow.GradientTape was used is shown in the appendix VI.

II. METHODS

A. The diffusion equation

The first equation we are going to solve is the diffusion equation which is a partial differential equation:

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \tag{1}$$

For our specific problem, we define

$$\begin{aligned} &x \in [0,1], \quad t > 0 \\ &u(x,0) = \sin \pi x \quad 0 < x < L \\ &u(0,t) = 0 \\ &u(L,t) = 0 \end{aligned} \tag{2}$$

These conditions model the temperature change in a rod with length $L = 1$, when it starts with a temperature 0 in each end, and a temperature $\sin \pi x$ for every $0 < x < L$. The ends at $x = 0$ and $x = L = 1$, is set to have a constant temperature of 0. $u(x,t)$ is then the temperature at position x on the rod at time t.

We assume that we, through a separation of variables, can write the solution to the equation on the form

$$u(x,t) = F(x)G(t) \tag{3}$$

which, inserted in the PDE results in

$$\frac{F''}{F} = \frac{G'}{G} \tag{4}$$

where the derivative is with respect to x on the left hand side and t on the right hand side.

Since the equation must hold for all x's and t's, the expressions on both sides must be equal to a constant, which we choose to be $-\lambda^2$.

This gives us the two expressions

$$\begin{aligned} F'' + \lambda^2 F = 0 \\ G' + \lambda^2 G = 0 \end{aligned} \tag{5}$$

with general solutions

$$F(x) = A\sin\lambda x + B\cos\lambda x$$
$$G(t) = Ce^{-\lambda^2 t} \tag{6}$$

Since we have the initial condition $u(x,0) = \sin\pi x$, the constants must be $A = 1, B = 0, C = 1$ and $\lambda = \pi$.

For our specific problem with the stated initial conditions, the analytical solution to the diffusion equation is therefore

$$u(x,t) = e^{-\pi^2 t}\sin\pi x \tag{7}$$

B. Explicit Scheme algorithm

One way of solving the differential equation is to numerically calculate the state of the system at a later time from the state of the system at the current time. This is called an explicit method, and we will in particular use the explicit scheme algorithm when solving the diffusion equation numerically.

In the explicit scheme, we use the following approximation for the second derivative

$$u_{xx}(x_i,t_j) \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \tag{8}$$

with a truncation error $\mathcal{O}(\Delta x^2)$.

We also use the following approximation for the time derivative

$$u_t(x_i,t_j) \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t} \tag{9}$$

Since we know the value of $u_{i,j} = u(x_i,t_j)$ for all x_i at the time $t_j = 0$ from the initial conditions, we can iteratively find the values $u_{i,j+1} = u(x_i, t_j + \Delta t)$ for all the subsequent time steps by putting equation 8 and 9 into equation 1. Rewriting the expression we then get

$$u_{i,j+1} = \alpha u_{i+1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i-1,j} \tag{10}$$

where $\alpha = \frac{\Delta t}{\Delta x^2}$.

All of the terms on the right hand side in equation 10 are known for $t_i = 0$, which means the left hand side gives all the values for $t = 0 + \Delta t$. These values can again be used to find the values at the next timestep and so on.

The explicit scheme can be written in terms of a matrix-vector multiplication:

$$U_{j+1} = AU_j$$
$$= \begin{bmatrix} (1-2\alpha) & \alpha & 0 & \dots 0 \\ \alpha & (1-2\alpha) & \alpha & \dots 0 \\ \dots & \dots & \dots & \alpha \\ 0 & 0\dots & \alpha & (1-2\alpha) \end{bmatrix} \begin{bmatrix} u_{1,j} \\ u_{2,j} \\ \dots \\ u_{n,j} \end{bmatrix} \tag{11}$$

so that the values U_{j+1} can be found by repeated matrix multiplications:

$$U_{j+1} = AU_j = \dots = A^{j+1}U_0 \tag{12}$$

Since we want U_j to approach a definite value, A needs to have a spectral radius less than 1. This gives the constraint for convergence: $\alpha = \frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}$ [1].

C. Neural Network with TensorFlow

The Universal Approximation Theorem states that a neural network can approximate any function at a single hidden layer along with one input and output layer to any given precision [2].

Since a partial differential equation completely describes the behavior of the function we are looking for, we will try to approximate the function using a neural network, and optimizing it using gradient descent so that it fulfills the partial differential equation.

The equation we wish to solve is the diffusion equation, equation 13. We will model its solution $u(x,t)$ using a neural network $N(x,t)$, but to ensure that the initial conditions and the boundary conditions are fulfilled, we must include some extra terms and factors. In total, we model $u(x,t)$ by the trial function $g_t(x,t)$

$$g_t(x,t) = (1-t)sin(\pi x) + tx(1-x)N(x,t) \tag{13}$$

$g_t(x,t)$ is chosen such that $g(x,0) = sin(\pi x)$, and $g(0,t) = g(1,t) = 0$. Note that we will model $g_t(x,t)$ only at the M discrete points (x_i,t_j).

The neural network $N(x,t)$ is initialized with random weights and biases, which means that $g_t(x,t)$ will most likely not fulfill equation 13 initially. The degree to which $g_t(x,t)$ does not fulfill equation 13 will be given by the loss function L

$$L = \frac{1}{M}\sum_{i,j}\left(\frac{\partial^2 g_t(x_i,t_j)}{\partial x^2} - \frac{\partial g_t(x_i,t_j)}{\partial t}\right)^2 \tag{14}$$

Clearly, if $L = 0$, $g_t(x,t)$ fulfills equation 13, and if L is close to 0, we have a good approximation. We therefore wish to minimize L.

We will minimize L by updating $N(x,t)$ using the Adam implementation of stochastic gradient descent in TensorFlow.

To do this, we first need to calculate some derivatives. First, the derivatives of $g_t(x,t)$ which are used in L, then the derivative of L itself with respect to the all the parameters in the neural network $N(x,t)$. The number of equations involved in these derivatives makes finding analytical expressions impractical.

We will therefore use automatic differentiation to compute the derivatives. Automatic differentiation uses the fact that all calculations done in a computer program can be broken down into simple operations and functions(addition, subtraction, multiplication, division, sin,

cos, exp, log, etc.). By recording all the simple operations done in a calculation, one can then repeatedly use the chain rule on these operations to compute any derivative.

The tf.GradientTape API in TensorFlow lets us record all operations in a computation, and then compute the derivative of that computation with respect to any ("observed") variable. This means that we can evaluate $g_t(x,t)$ inside of a tf.GradientTape, and take its derivative wrt. x and t to evaluate the loss function L. We can then take this one step further and perform this calculation of L inside of another tf.GradientTape, to find the derivative of L wrt. the trainable variables in our neural network.

Code showing how tf.GradientTape is used to implement the loss function L, and how tf.GradientTape is used to update the model to minimize L is in the appendix (VI).

D. The eigenvalue problem

The method discussed in the previous section of fitting a neural network to a differential equation is very flexible, and can be used in other contexts as well. What follows are the central results in a method for finding the eigenvalues of a matrix discussed in [3].

Let A be a $n \times n$ real symmetric matrix. Let $x(t)$ denote a vector in R^n at time $t \geq 0$ which follows the equation

$$\frac{dx(t)}{dt} = -x(t) + f(x(t)) \tag{15}$$

where

$$f(x) = [x^T x A + (1 - x^T A x)I]x \tag{16}$$

and where I is the $n \times n$ identity matrix.

$\lim_{t\to\infty} x(t)$ converges to the eigenspace of A with the largest eigenvalue, among the eigenspaces which are not orthogonal to the starting point $x(0)$.

If $x(t)$ follows equation 15 with the matrix $-A$ instead, $\lim_{t\to\infty} x(t)$ converges to the eigenspace of A with the smallest eigenvalue, among the eigenspaces which are not orthogonal to $x(0)$.

This gives us a method for finding the largest and smallest eigenvalues of a real symmetric matrix. We let a neural network model the vector $x(t)$, with the initial condition of starting at a vector $x(0)$ at time 0. One way to do this is to model $x(t)$ with the trial function $x_t(t)$

$$x_t(t) = e^{-t}x(0) + (1 - e^{-t})N(t) \tag{17}$$

where $N(t)$ is a neural network which outputs a vector in R^n. Note that we will model $x_t(t)$ only at the M discrete times t_i. We use equation 15 to define the loss function L:

$$L = \frac{1}{M}\sum_i \left(\frac{dx_t(t_i)}{dt} + x_t(t_i) - f(x_t(t_i))\right)^2 \tag{18}$$

Just like the previous case, if L is close to 0, $x_t(t)$ is a good approximation to the solution of equation 15, $x(t)$, and $\lim_{t\to\infty} x_t(t)$ is close to an eigenvector of A. We therefore wish to minimize L.

To do this, we use automatic differentiation to calculate $dx_t(t_i)/dt$, and to calculate the derivative of L wrt. the parameters of the neural network $N(t)$. We then use these derivatives to update the network using the Adam implementation of stochastic gradient descent in TensorFlow.

To judge how close $x(t)$ converges to an eigenvector, we extract the value of $x(t)$ at a time when it seems to have converged, $x(t_c)$. We then compute $Ax(t_c)$ and divide element-wise by $x(t_c)$ to see how much each element was scaled. The mean of these scalings will be the eigenvalue approximation, and the standard deviation of these scalings will tell us how close $x(t_c)$ is to being an eigenvector.

All eigenvalue-results in this report were computed with 2000 iterations of Adam optimization, with 11 time steps from 0 to 10 unless otherwise specified. The neural network consists of three hidden layers with sigmoid activation and 100, 50 and 25 nodes. The output layer has as many nodes as the dimension of the matrix we are using, and linear activation. The network structure was chosen as it performed well, and we only include results from this network structure as we prioritize showing how the matrix size and step length affect the results of the method.

III. RESULTS

A. The diffusion equation

We tested how well both the explicit scheme algorithm and the neural network managed to solve the diffusion equation. The methods were evaluated against the analytical solution. We tested the methods for $\Delta x = 1/10$ and $\Delta x = 1/100$, training the neural network with 10 000 iterations in both cases. We used $\Delta t = \frac{\Delta x^2}{2}$ in our calculations.

We show the results with 2D plots for the case of $\Delta x = 1/10$ in figure 1 and with 3D plots when $\Delta x = 1/100$ in figure 2. More figures can be found in the file "Diffusion.ipynb" in the Github repository [4].

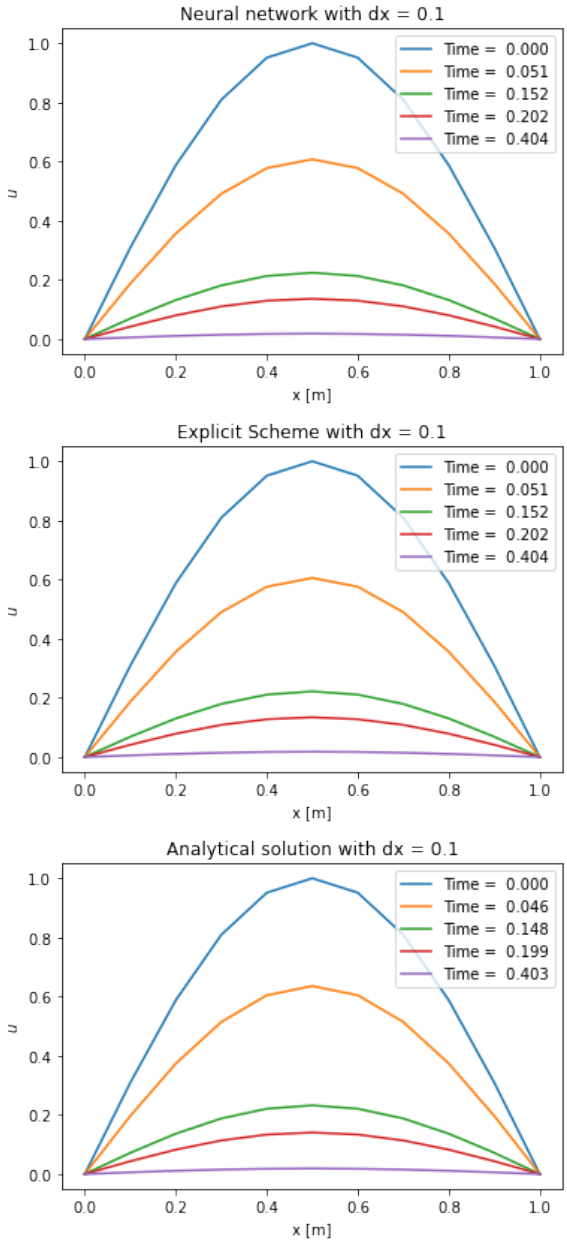

Figure 1. 2D plots of the different methods solving the diffusion equation for $\Delta x = 1/10$

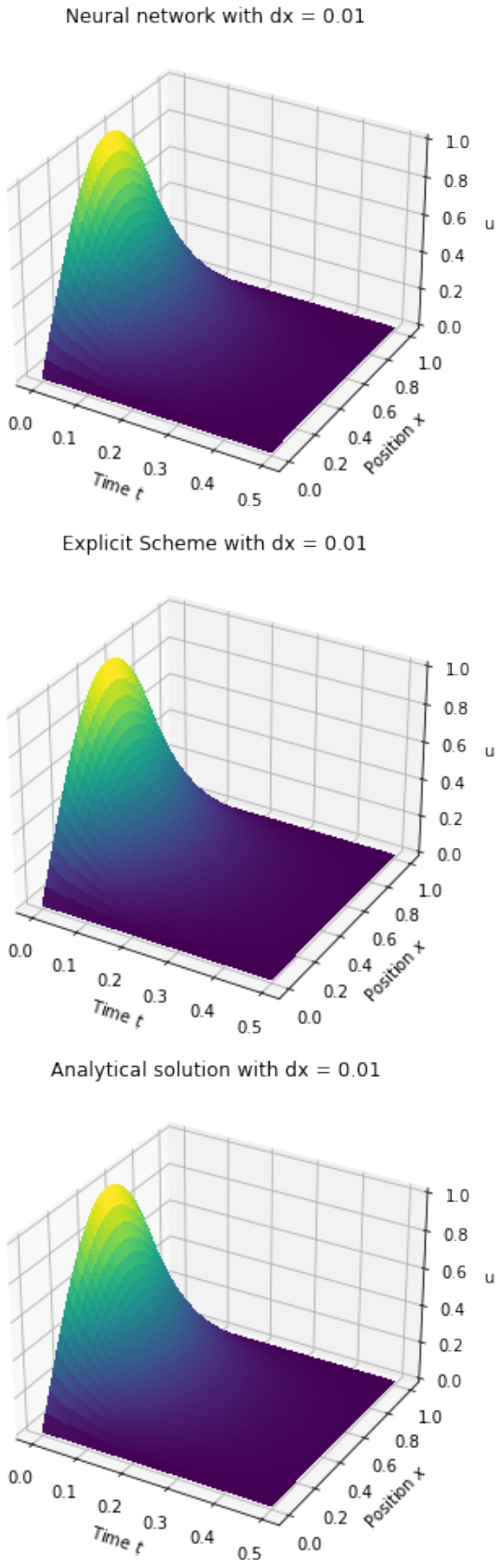

Figure 2. 3D plots of the different methods solving the diffusion equation for $\Delta x = 1/100$

The plots for the different methods are quite alike and not easily differentiated. Table I shows the mean and max absolute error of the explicit scheme algorithm and the neural network compared to the analytical solution.

Method	Δx	Mean absolute error	Max absolute error
ES	0.1	7.3752e-04	2.4663e-03
NN	0.1	1.5866e-04	1.0476e-03
ES	0.01	2.0113e-05	6.0525e-05
NN	0.01	1.5590e-03	4.3186e-03

Table I. Performance of the neural network (NN) and the explicit scheme (ES) measured against the analytical solution for different Δx.

As explained in part II B, the explicit scheme requires that $\Delta t \leq \frac{1}{2}\Delta x^2$, and figure 3 shows what happens if we ignore this stability criterion.

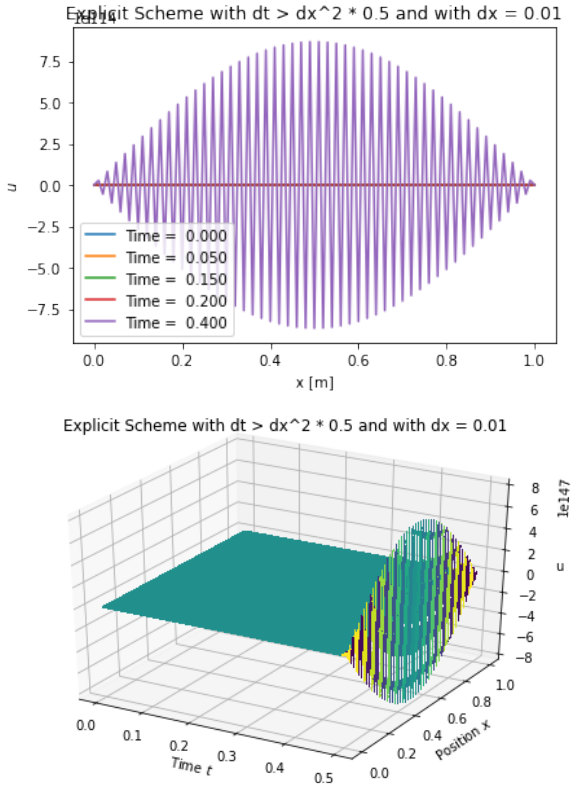

Figure 3. 2D- and 3D plot for the explicit scheme algorithm when $\Delta t > \frac{1}{2}\Delta x^2$ (more precisely, is this case, $\Delta t = 0.51\Delta x^2$).

B. The eigenvalue problem

We used a neural network to solve equation 15 with a random matrix A and a random starting vector $x(0)$.

Figure 4 shows the components of $x(t)$ when the equation was solved with the matrix and initial vector

$$A = \begin{bmatrix} 0.048 & 0.295 & 0.427 & 0.390 & 0.559 & 0.188 \\ 0.295 & 0.663 & 0.243 & 0.483 & 0.415 & 0.663 \\ 0.427 & 0.243 & 0.495 & 0.636 & 0.637 & 0.757 \\ 0.390 & 0.483 & 0.636 & 0.713 & 0.219 & 0.535 \\ 0.559 & 0.415 & 0.637 & 0.219 & 0.298 & 0.541 \\ 0.188 & 0.663 & 0.757 & 0.535 & 0.541 & 0.726 \end{bmatrix}$$

$$x(0) = \begin{bmatrix} 0.849 \\ 0.267 \\ 0.614 \\ 0.652 \\ 0.386 \\ 0.410 \end{bmatrix}$$

The diamonds at the end show the components of the actual eigenvector, computed with the standard python library numpy.

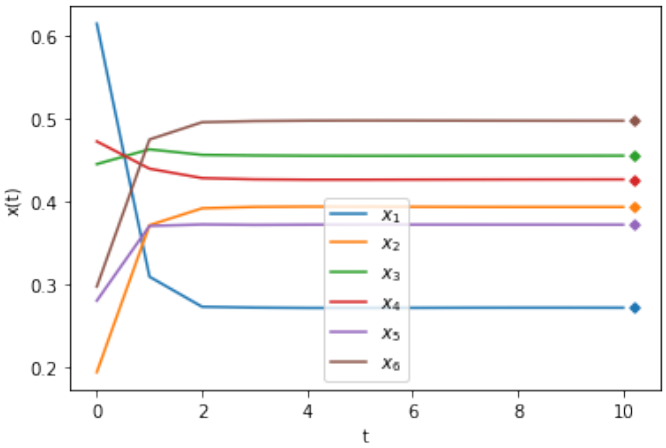

Figure 4. The solution to equation 15 found using a neural network. A random symmetric 6×6 matrix A and random start vector $x(0)$ was used. The diamonds on the right show the components of the eigenvector of A with the largest eigenvalue. $x(t)$ was normalized at each step in the plot.

The eigenvector corresponding to the largest eigenvalue of A, and $x(10)$ (which is our approximation of the eigenvector) are:

$$v_{\lambda max} = \begin{bmatrix} 0.2720 \\ 0.3925 \\ 0.4541 \\ 0.4251 \\ 0.3709 \\ 0.4974 \end{bmatrix}, x(10) = \begin{bmatrix} 0.2711 \\ 0.3926 \\ 0.4545 \\ 0.4259 \\ 0.3712 \\ 0.4966 \end{bmatrix}$$

Figure 5 shows the components of $x(t)$ when the equation was solved with $-A$ instead of A. The third and sixth element of the actual eigenvector overlap (green and brown).

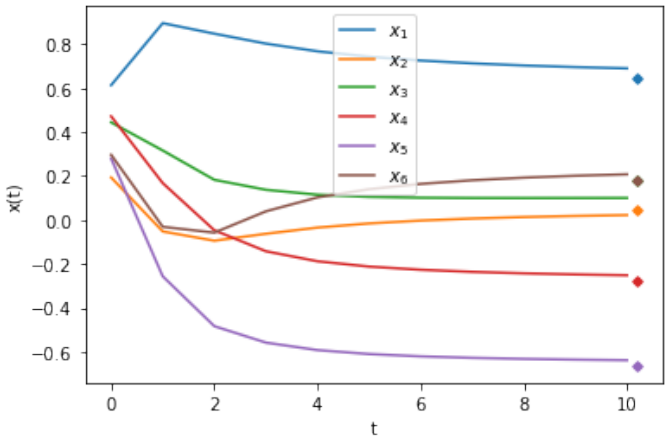

Figure 5. The solution to equation 15 found using a neural network. A random symmetric 6×6 matrix A and random start vector $x(0)$ was used. The diamonds on the right show the components of the eigenvector of A with the smallest eigenvalue. $x(t)$ was normalized at each step in the plot.

Figure 5 shows the components of $x(t)$ when the equation was solved with A and $x(0)$, but with 51 time steps.

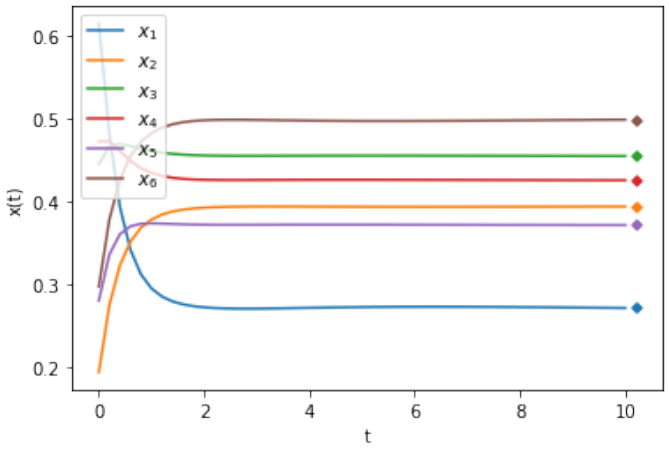

Figure 6. The solution to equation 15 found using a neural network. A random symmetric 6×6 matrix A and random start vector $x(0)$ was used. The diamonds on the right show the components of the eigenvector of A with the largest eigenvalue. This simulation used 51 time steps. $x(t)$ was normalized at each step in the plot.

The eigenvalues of A are shown in table II.

Eigenvalues
2.90759
-0.50203
-0.40458
0.18207
0.42070
0.34353

Table II. The eigenvalues of the 6×6 matrix A we use to test our neural network.

How the neural network performed in finding the largest eigenvalue of A for different step sizes in time is shown in table III.

Δ t	λ	Error from actual
2	2.90757 $\pm$ 0.00110	-0.00003
1	2.90792 $\pm$ 0.00220	0.00032
0.2	2.90899 $\pm$ 0.00579	0.00140
0.1	2.90999 $\pm$ 0.02202	0.00239

Table III. The eigenvalues found by solving equation 15 with a neural network and letting $x(10)$ be the "eigenvector". Results shown for different step sizes in time. The mean and standard deviation of the scaling of the components of $x(10)$ when multiplying with the matrix A used in the equation gives our approximation λ.

Results for a 3×3, 9×9 and 18×18 matrix are shown in figures 7, 8 and 9.

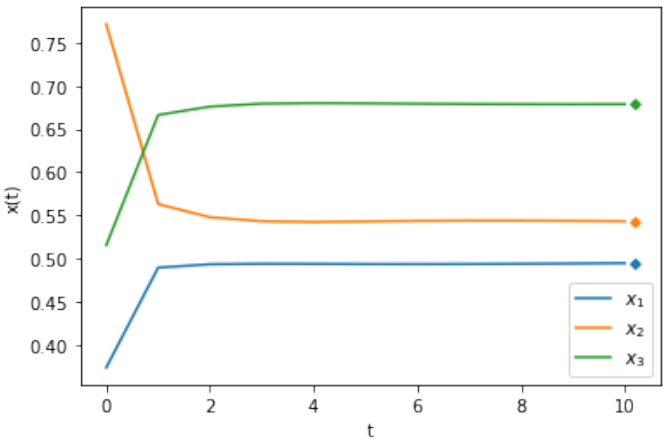

Figure 7. The solution to equation 15 found using a neural network. A random symmetric 3×3 matrix A and random start vector $x(0)$ was used. The diamonds on the right show the components of the eigenvector of A with the largest eigenvalue. $x(t)$ was normalized at each step in the plot.

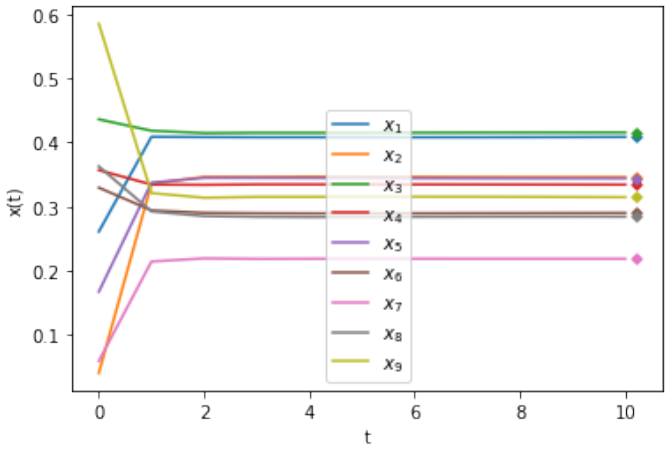

Figure 8. The solution to equation 15 found using a neural network. A random symmetric 9×9 matrix A and random start vector $x(0)$ was used. The diamonds on the right show the components of the eigenvector of A with the largest eigenvalue. $x(t)$ was normalized at each step in the plot.

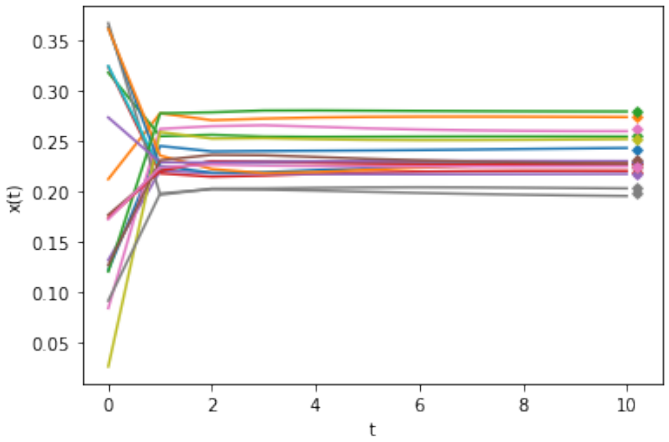

Figure 9. The solution to equation 15 found using a neural network. A random symmetric 18×18 matrix A and random start vector $x(0)$ was used. The diamonds on the right show the components of the eigenvector of A with the largest eigenvalue. $x(t)$ was normalized at each step in the plot.

Table IV shows how well the neural network found the largest and smallest eigenvalues of a 3×3, 6×6, 9×9 and 18×18 matrix (the same matrices as the previous results).

N	± A(large-/smallest)	λ	Error from actual
3	+	1.95691 ± 0.00152	-0.00026
3	-	-0.23605 ± 0.00912	0.00663
6	+	2.90792 ± 0.00220	0.00032
6	-	-0.47934 ± 0.11662	0.02269
9	+	4.51152 ± 0.00302	0.00039
9	-	-0.87007 ± 0.02441	0.00832
18	+	8.96900 ± 0.08441	0.00117
18	-	-1.63169 ± 0.70949	0.02061

Table IV. How well the neural network was able to approximate the largest (+) and smallest (-) eigenvalues of a $N \times N$ matrix. The eigenvalues were found by solving equation 15 with a neural network and letting $x(10)$ be the "eigenvector". The mean and standard deviation of the scaling of the components of $x(10)$ when multiplying with the matrix A used in the equation gives our approximation λ.

The network either diverged to infinity or converged too slowly when using stochastic gradient descent with a fixed learning rate. Adam optimization with its adaptive learning rate was chosen as it gave reliable results. To be safe, we tested 4 more 6×6 matrices thoroughly and got results which reinforced the behavior we saw in the example we presented in this report. Every calculation took around 15 minutes, regardless of matrix size or step length. All of these results can be found in our repository [4].

IV. DISCUSSION

A. The diffusion equation

The neural network was able to solve the diffusion equation and even outperform the explicit scheme for large Δx and Δt. But for smaller step sizes, the neural network's performance worsened, while the explicit schemes of course improved greatly due to the smaller truncation error.

The neural network performed best for large step sizes, as it has the ability to interpolate, and to better follow the derivatives at each time step. This is however not a big advantage as the expense of training the network is much larger than the expense of using a traditional method with a smaller step size.

We are unsure of why the network performed worse with a smaller step size. It might be due to our network being unable to fit the large number of data points due to having too few nodes.

B. The eigenvalue problem

We were able to approximate the largest and smallest eigenvalues of a symmetric matrix by solving equation 15 with a neural network.

The method has however some limitations. A 15 minute calculation for an error of 3E-5 at best means

this method used on this problem has few practical uses. As seen in figure 5 and other simulations [4], when the eigenvalue the algorithm is looking for is close to other eigenvalues, $x(t)$ converges both slowly, and to the wrong vector. In the case of figure 5, the smallest eigenvalue -0.502 is very close to the second smallest eigenvalue -0.405, which makes $x(t)$ converge to the wrong vector.

The fact that the method was better at approximating the largest eigenvalue than the smallest eigenvalue is because our method of generating matrices leads to the largest eigenvalue being much larger than the others, while the smallest eigenvalue often is close to other eigenvalues.

The error in our eigenvalue approximation was consistently much smaller than the standard deviation of the scaling of the elements in our eigenvector approximation. This is because the elements which were scaled too much were canceled out by elements which were scaled too little when we took the mean of the scalings. The consistency of this result shows that the eigenvalue approximation is much more accurate than the standard deviation of the scalings shows.

The algorithm also performed better with larger step sizes in time. This was surprising, as we expected a smaller step size to better capture the behavior described by equation 15, and thus give a better result for the eigenvector. One theory we have for this behavior is that the large step size gives the neural network fewer points before convergence to fit, which lets it "focus" more on $x(t)$ at points near convergence.

Smaller matrices gave better results, which is expected due to the equation to solve being much simpler for smaller matrices. Increasing the matrix size from 3×3 to 6×6, 9×9 and 18×18 resulted however in only a small increase in error, which speaks for the strength and flexibility of the neural network.

V. CONCLUSION

We have shown that neural networks are able to solve partial differential equations with the help of automatic differentiation. However, in the two problems we looked at, the diffusion equation and the eigenvalue problem, traditional methods were shown to be far superior.

For the diffusion equation, the neural network was able to outperform the explicit scheme when the step size was large, due to its ability to interpolate better. But for smaller step sizes, the explicit scheme was able to improve greatly due to a smaller truncation error, while the neural network was both slow to train and even performed worse than with the large step size. The explicit scheme easily achieved a mean absolute error of $2.0113e-05$, while the neural network only achieved a mean absolute error of $1.5866e-04$.

For the eigenvalue problem, the neural network was very sensitive to the specific matrix it was finding the eigenvalues of. The neural network could only find the largest or smallest eigenvalue, and if there were other eigenvalues close to these values, the network struggled to converge to the correct value. Even for this problem, the network performed worse with a larger step size. It took around 15 minutes of calculations to achieve an error of $3e-05$, but the most typical result had an error closer to $1e-04$. This is of course very slow and imprecise compared to typical eigenvalue solvers, which find the eigenvalues of such small matrices with much greater accuracy nearly instantly.

However, our results are more a proof of the flexibility of neural networks than a rigorous test of their efficiency at solving these problems. Future improvements to our testing can include testing out different types of network structures and types of networks, like recurrent neural networks. Seeing if a network trained on finding the eigenvalues of a matrix is faster to train on a different matrix. And of course looking more into the effect of step length on the neural networks.

[1] M. Hjorth-Jensen, *Computational Physics Lecture Notes 2015* (Department of Physics, University of Oslo, Norway, 2015) pp. 301–308.

[2] K. B. Hein, *Data Analysis and Machine Learning: Using Neural networks to solve ODEs and PDEs* (Department of Informatics, University of Oslo, Norway, 2018) pp. 1–41.

[3] H. J. T. Zhang Yi, Yan Fu, Computers Mathematics with Applications **47**, 1155 (2004).

[4] "Github repository with code and results: https://github.com/KarlHenrik/FYS-STK-Gruppe/tree/master/Project3," (15.12.2020).

VI. APPENDIX

The (psuedo)code to calculate the derivatives of $g_t(x,t)$ used in the loss function L. The derivatices are calculated using the "tf.GradientTape" API in TensorFlow.

```
def loss(model, x, t):
    with tf.GradientTape() as tape_x2:
        tape_x2.watch([x])
        with tf.GradientTape() as tape_x, tf.GradientTape() as tape_t:
            tape_x.watch([x])
            tape_t.watch([t])
            g_trial = (1 - t) * sin(pi * x) + x * (1 - x) * t * N(x,t])

        dg_dx = tape_x.gradient(g_trial, x)
        dg_dt = tape_t.gradient(g_trial, t)

    dg_d2x = tape_x2.gradient(dg_dx, x)

    return tf.losses.mean_squared_error(zeros, dg_d2x - dg_dt)
```

The code to update the model by taking the derivative of the loss function wrt. the trainable variables in the neural network. This code is run inside a loop, as it takes many iterations to train the model. Notice how there are three levels of automatic differentiation, since the loss function is called inside of a tf.GradientTape.

```
with tf.GradientTape() as tape:
    current_loss = loss(model, x, t)

grads = tape.gradient(current_loss, model.trainable_variables)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```